HW1

(6 min.) Write a function named `count_occurrences` that returns the number of ways that all elements of list a_1 appear in list a_2 in the same order (though a_1's items need not necessarily be consecutive in a_2). The empty sequence appears in another sequence of length `n` in 1 way, even if `n` is 0.

Examples:

`count_occurrences [10, 20, 40] [10, 50, 40, 20, 50, 40, 30]` should return 1.

`count_occurrences [10, 40, 30] [10, 50, 40, 20, 50, 40, 30]` should return 2.

`count_occurrences [20, 10, 40] [10, 50, 40, 20, 50, 40, 30]` should return `0`.

`count_occurrences [50, 40, 30] [10, 50, 40, 20, 50, 40, 30]` should return 3.

`count_occurrences [] [10, 50, 40, 20, 50, 40, 30]`

should return 1.

`count_occurrences [] []` should return 1.

`count_occurrences [5] []` should return `0`.

Solution:

```haskell
count_occurrences :: [Integer] -> [Integer] -> Integer
count_occurrences [] _ = 1
count_occurrences _ [] = 0
count_occurrences (x:xs) (y:ys)
  | x == y = count_occurrences xs ys + other_occurrences
  | otherwise = other_occurrences
  where other_occurrences = count_occurrences (x:xs) ys
```

HW2

(2 min.) Use the `map` function to write a Haskell function named `scale_nums` that takes in a list of `Integers` and an `Integer` named `factor`. It should return a new list where every number in the input list has been multiplied by `factor`.

Example:

`scale_nums [1, 4, 9, 10] 3` should return `[3, 12, 27, 30]`.

You may not define any nested functions. Your solution should be a single, one-line map expression that includes a lambda.

Solution:

```haskell
scale_nums :: [Integer] -> Integer -> [Integer]
scale_nums xs factor = map (\x -> x * factor) xs
```

(2 min.) Use the `filter` and `all` functions to write a Haskell function named `only_odds` that takes in a list of `Integer` lists, and returns all lists in the input list that only contain odd numbers (in the same order as they appear in the input list). Note that the empty list vacuously satisfies this requirement.

Example:

`only_odds [[1, 2, 3], [3, 5], [], [8, 10], [11]]` should return `[[3, 5], [], [11]]`.

You may not define any nested functions. Your solution should be a single, one-line `filter` expression that includes a lambda.

Solution:

```haskell
-- note the partial application with all
only_odds :: [[Integer]] -> [[Integer]]
only_odds xs = filter (all (\x -> mod x 2 /= 0)) xs
```

a) In Homework 1, you wrote a `largest` function that returns the larger of two words, or the first if they are the same length:

```
largest :: String -> String -> String
largest first second =
    if length first >= length second then first else second
```

Use one of `foldl` or `foldr` and the `largest` function to write a Haskell function named `largest_in_list` that takes in a list of `String`s and returns the longest `String` in the list. If the list is empty, return the empty string. If there are multiple strings with the same maximum length, return the one that appears first in the list. **Do not use the map, `filter` or `maximum` functions in your answer.**

Your answer should be a single, one-line `fold` expression.

Example:

`largest_in_list ["how", "now", "brown", "cow"]` should return `"brown"`.

`largest_in_list ["cat", "mat", "bat"]` should return `"cat"`

Solution:

```
largest_in_list :: [String] -> String
largest_in_list xs = foldl largest "" xs
```

1.

a) (5 min.) Write a Haskell function named `count_if` that takes in a predicate function of type `(a -> Bool)` and a list of type `[a]`. It should return an `Int` representing the number of elements in the list that satisfy the predicate. **Your solution must use recursion. Do not use the map, `filter`, `foldl`, or `foldr` functions in your solution.**

Examples:

`count_if (\x -> mod x 2 == 0) [2, 4, 6, 8, 9]` should return `4`.

`count_if (\x -> length x > 2) [“a”, “ab”, “abc”]` should return `1`.

Solution:

```
count_if :: (a -> Bool) -> [a] -> Int
count_if predicate [] = 0
count_if predicate (x:xs)
    | (predicate x) = 1 + count_if predicate xs
    | otherwise = count_if predicate xs
```

b) (3 min.) Now, reimplement the same function above (call it `count_if_with_filter`), **but use the `filter` function in your solution**.

Solution:

```
count_if_with_filter :: (a -> Bool) -> [a] -> Int
count_if_with_filter predicate xs = length (filter predicate xs)
```

c.)

Now, reimplement the same function above (call it `count_if_with_fold`) **but use either `foldl` or `foldr` in your solution**.

Solution:

```
count_if_with_fold :: (a -> Bool) -> [a] -> Int
count_if_with_fold predicate xs =
  let count acc x = if predicate x then acc + 1 else acc
```

```haskell
  in foldl count 0 xs
```

1. Consider the following Haskell function:

```haskell
f a b =
  let c = \a -> a    -- (1)
      d = \c -> b    -- (2)
  in \e f -> c d e   -- (3)
```

a) (1 min.) What variables (if any) are captured in the lambda labeled `(1)`?

There are no captured variables. The parameter to the lambda (a in `\a`) "shadows" (hides) the outer a parameter that's passed into `f`. So the outer a is not captured. The line is effectively equivalent to "c = \x -> x" (or any other name for the bound variable).

If it is in some form of an assignment, then the variable is not captured, also c was just declared, so it can't be captured. It has to exist in some other outer variable scope in order for it to be captured

b) (1 min.) What variables (if any) are captured in the lambda labeled `(2)`?

b is captured (the second parameter of `f`). c is not captured; the line is effectively equivalent to "d = \y -> b". Same thing here

c) (1 min.) What variables (if any) are captured in the lambda labeled `(3)`?

c and d are captured (from the `let` declarations). We can understand this as c,d being replaceable with the implementations defined by the `let` declarations. Note that the f in this line has nothing to do with the name of the function being f.

d) (2 min.) Define a new Haskell type `InstagramUser` that has two value constructors (without parameters) - `Influencer` and `Normie`.

Solution:

```haskell
data InstagramUser = Influencer | Normie
```

e) (2 min.) Write a function named `lit_collab` that takes in two `InstagramUsers` and returns `True` if they are both `Influencers` and `False` otherwise.

Solution:

```haskell
lit_collab :: InstagramUser -> InstagramUser -> Bool
lit_collab Influencer Influencer = True
lit_collab _ _ = False
```

Consider the following code example:

```haskell
data Character = Hylian Int | Goron | Rito Double | Gerudo | Zora

describe :: Character -> String
describe (Hylian age) = "A Hylian of age " ++ show age
describe Goron = "A Goron miner"
describe (Rito wingspan) = "A Rito with a wingspan of " ++ show wingspan ++ "m"
describe Gerudo = "A mighty Gerudo warrior"
describe Zora = "A Zora fisher"
```

f) (2 min.) Modify your `InstagramUser` type so that the `Influencer` value constructor takes in a list of Strings representing their sponsorships.

Solution:

```haskell
data InstagramUser = Influencer [String] | Normie
```

g) (3 min.) Write a function `is_sponsor` that takes in an `InstagramUser` and a `String` representing a sponsor, then returns `True` if the user is sponsored by `sponsor` (this function always returns `False` for `Normies`).

Solution:

```haskell
is_sponsor :: InstagramUser -> String -> Bool
is_sponsor Normie _ = False
is_sponsor (Influencer sponsors) sponsor =
  sponsor `elem` sponsors
```

Consider the following code example:

```haskell
data Quest = Subquest Quest | FinalBoss

count_subquests :: Quest -> Integer
count_subquests FinalBoss = 0
count_subquests (Subquest quest) = 1 + count_subquests quest
```

h) (2 min.) Modify your `InstagramUser` type so that the `Influencer` value constructor also takes in a list of other `InstagramUsers` representing their followers (after their sponsors).

Solution:

```haskell
data InstagramUser = Influencer [String] [InstagramUser] | Normie
```

i) (3 min.) Write a function `count_influencers` that takes in an `InstagramUser` and returns an `Integer` representing the number of `Influencers` that are following that user (this function always returns `0` for `Normies`).

Solutions:

```
-- foldl-based solution
count_influencers :: InstagramUser -> Integer
count_influencers Normie = 0
count_influencers (Influencer _ followers) =
  let count acc (Influencer _ _) = acc + 1
      count acc Normie = acc
  in foldl count 0 followers
```

```
-- hand-coded recursive solution
count_influencers:: InstagramUser -> Integer
count_influencers Normie = 0
count_influencers (Influencer _ followers) =
  let count ((Influencer s f):xs) = 1 + count xs
      count (_:xs) = count xs
      count [] = 0
  in count followers
```

j) (2 min.) Use GHCi to determine the type of `Influencer` using the command `:t Influencer`. What can you infer about the type of custom value constructors?

Value constructors are just functions that return an instance of the custom data type!

k) ** (10 min.) Using Haskell, write a function named `longest_run` that takes in a list of `Bool`s and returns the length of the longest consecutive sequence of `True` values in that list.

Examples:

`longest_run [True, True, False, True, True, True, False]` should return `3`.

`longest_run [True, False, True, True]` should return `2`.

Solution using helper parameters:

```
longest_run :: [Bool] -> Int
longest_run xs =
  let
    longest_run_helper [] current_run current_max =
      max current_run current_max
    longest_run_helper (x:xs) current_run current_max =
      if x
        then longest_run_helper xs (current_run+1) current_max
        else longest_run_helper xs 0 (max current_run current_max)
  in
    longest_run_helper xs 0 0
```

a) ** (5 min.) Consider the following Haskell data type:

```
data Tree = Empty | Node Integer [Tree]
```

Using Haskell, write a function named `max_tree_value` that takes in a `Tree` and returns the largest `Integer` in the `Tree`. Assume that all values in the tree are non-negative. If the root is `Empty`, return `0`.

Example:

`max_tree_value (Node 3 [(Node 2 [Node 7 []]), (Node 5 [Node 4 []])])` should return 7.

```
-- map-based solution
max_tree_value :: Tree -> Integer
max_tree_value Empty = 0
max_tree_value (Node value children) =
  case children of
    [] -> value
    xs -> max value (maximum (map max_tree_value xs))
```

9. Write a Haskell function named `fibonacci` that takes in an `Int` argument n. It should return the first n numbers of the Fibonacci sequence (for this problem, we'll say that the first two numbers of the sequence are 1, 1).

   Examples:

   `fibonacci 10` should return `[1,1,2,3,5,8,13,21,34,55].`

   `fibonacci -1` should return `[].`

   Hint: You may find it easier to build the list in reverse in a right-to-left manner, then use the reverse function.

   Solutions:

```
-- solution where we build the list in reverse
fibonacci :: Int -> [Integer]
fibonacci n =
  let fib_rev 1 = [1]
      fib_rev 2 = [1, 1]
      fib_rev n =
        let prev_fib_rev = fib_rev (n-1)
            first = head prev_fib_rev
            second = head (tail prev_fib_rev)
        in (first + second) : prev_fib_rev
  in reverse (fib_rev n)
```

9. (20 min.) Super Giuseppe is a hero trying to save Princess Watermelon from the clutches of the evil villain Oogway. He has a long journey ahead of him, which is comprised of many events:

```
data Event = Travel Integer | Fight Integer | Heal Integer
```

   Super Giuseppe begins his adventure with 100 hit points, and may never exceed that amount. When he encounters:

   A `Travel` event, the `Integer` represents the distance he needs to travel. During this time, he heals for ¼ of the distance traveled (floor division).

   A `Fight` event, the `Integer` represents the amount of hit points he loses from the fight.

   A `Heal` event, the `Integer` represents the amount of hit points he heals after consuming his favorite power-up, the bittermelon.

   If Super Giuseppe has 40 or fewer life points after an Event, he enters defensive mode. While in this mode, he takes half the damage from fights (floor division), but he no longer heals while traveling. Nothing changes with Heal events. Once he heals **above** the 40 point threshold following an Event, he returns to his normal mode (as described above).

   Write a Haskell function named `super_giuseppe` that takes in a list of `Events` that comprise Super Giuseppe's journey, and returns the number of hit points that he has at the end of it. If his hit points hit 0 or below at any point, then he has unfortunately Game Over-ed and the function should return -1.

Examples:

`super_giuseppe [Heal 20, Fight 20, Travel 40, Fight 60, Travel 80, Heal 30, Fight 40, Fight 20]` should return `10`.

`super_giuseppe [Heal 40, Fight 70, Travel 100, Fight 60, Heal 40]` should return `-1`.

Solutions:

```haskell
-- applying each event recursively
super_giuseppe xs =
  let normal_step (Travel distance) hp =
        min 100 (hp + distance `div` 4)
      normal_step (Fight loss) hp = hp - loss
      normal_step (Heal amount) hp = min 100 (hp + amount)
      defensive_step (Travel _) hp = hp
      defensive_step (Fight loss) hp = hp - (loss `div` 2)
      defensive_step (Heal amount) hp = min 100 (hp + amount)
      stepper [] hp = hp
      stepper (x:xs) hp =
        let new_hp = if hp <= 40 then defensive_step x hp
                                 else normal_step x hp
        in if new_hp <= 0 then (-1) else stepper xs new_hp
  in stepper xs 100
```

HW3: ** Consider the following Haskell data type:

```haskell
data LinkedList = EmptyList | ListNode Integer LinkedList
  deriving Show
```

a. ** (3 min.) Write a function named `ll_contains` that takes in a `LinkedList` and an `Integer` and returns a `Bool` indicating whether or not the list contains that value.

   Examples:

   `ll_contains (ListNode 3 (ListNode 6 EmptyList)) 3`

   should return `True`.

   `ll_contains (ListNode 3 (ListNode 6 EmptyList)) 4`

   should return `False`.

Solution:

```haskell
ll_contains :: LinkedList -> Integer -> Bool
ll_contains EmptyList _ = False
ll_contains (ListNode x next) y =
  if x == y then True else ll_contains next y
```

b. ** (3 min.) We want to write a function named `ll_insert` that inserts a value at a given zero-based index into an existing `LinkedList`. Provide a type definition for this function, explaining what each parameter is and justifying the return type you chose.

Solution:

```
ll_insert :: LinkedList -> Integer -> Integer -> LinkedList
```

Given the definition of the `LinkedList` data type, there is no way for us to modify a `LinkedList` passed to `ll_insert`. Therefore, what makes most sense is for `ll_insert` to return a new `LinkedList` where the value has been inserted at the desired index. The first parameter of our function will be the `LinkedList` to insert into, the second the value, and the third the index.

c. ** (5 min.) Implement the `ll_insert` function. If the insertion index is 0 or negative, insert the value at the beginning. If it exceeds the length of the list, insert the value at the end. Otherwise, the value should have the passed-in insertion index after the function is invoked.

Solution:

```
ll_insert :: LinkedList -> Integer -> Integer -> LinkedList
ll_insert EmptyList x _ = ListNode x EmptyList
ll_insert (ListNode y rest) x index
  | index <= 0 = ListNode x (ListNode y rest)
  | otherwise = ListNode y (ll_insert rest x (index-1))
```

5. (10 min) Inside and Outside the Shadow Realm [Scoping - Shadowing]

Suppose we have the following class:

```python
outside = 100

class Inside:
    def __init__(self):
        self.outside = outside

    def change_outside_1(self):
        self.outside += 10

    def change_outside_2(self):
        global outside
        outside += 20

    def change_outside_3(self):
        outside = 30

    def change_outside_4(self):
        def change_outside_5(outside):
            outside += 50
        global outside
        outside += 40
        change_outside_5(outside)
```

```python
a = Inside()
print("Initial values:")
print("global:", outside)
print("instance:", a.outside)
a.change_outside_1()
print("After 1:")
print("global:", outside)
print("instance:", a.outside)
a.change_outside_2()
print("After 2:")
print("global:", outside)
print("instance:", a.outside)
a.change_outside_3()
print("After 3:")
print("global:", outside)
print("instance:", a.outside)
a.change_outside_4()
print("After 4:")
print("global:", outside)
print("instance:", a.outside)
```

Initializing an Inside class creates an instance variable called “outside”, which is initialized to the global outside value, 100. Note that reading a global value does not require the global keyword, only changing or initializing.

change_outside_1 uses self.outside to access the instance variable, so only the instance variable is changed.

change_outside_2 uses the global keyword to tell Python that we want to modify the global outside variable, so the global variable is changed.

change_outside_3 creates a new local variable named outside, since it is neither global nor the self.outside instance of the class.

change_outside_4 uses the global outside, and changes it. It then calls change_outside_5 with a parameter also called outside. This means that inside the change_outside_5 function, “outside” refers to the parameter (which is treated as a local variable) and nothing else. Since Python passes by object reference, changing what “outside” is assigned to within the function doesn’t change it outside the function scope. Thus, the global outside is unchanged by the change_outside_5 call.

The values are as follows:

Time	**Global “outside”**	**Instance variable “outside”**
On initialization	100	100
After change_outside_1	100	110
After change_outside_2	120	110
After change_outside_3	120	110
After change_outside_4	160	110

8. (15 min) Tangled in Types [Midterm Review - Haskell Type Annotations]

For each of the following functions, determine a generic appropriate type annotation.

a) (5 min)

```
mysteryFunc1 c bar = filter (\x -> head x == c) bar
```

`mysteryFunc1 :: a -> [[a]] -> [[a]]`

This seems to be a function that takes in an element of type `a` (aka `c`) and a list of lists of type `a` (aka `bar`), then filters the elements in `bar` starting with `c`

b) (5 min)

```
mysteryFunc2  lists = foldr (++) [] lists
```

`mysteryFunc2 :: [[a]] -> [a]`

This is a function that takes in a list of lists and flattens it into a single list of elements utilizing `foldr`

c) (2 min)

```
mysteryFunc3 (x, y) = (y, x)
```

`mysteryFunc3 :: (a, b) -> (b, a)`

This function swaps the elements in a tuple of two different types.

d) (3 min)

```
data PasswordStrength = Awful | Weak | Strong | VeryStrong

mysteryFunc4 password
    | password == “” = Awful
    | length password < 8 = Weak
    | any isUpper password && any isLower password && any isDigit password = VeryStrong
    | otherwise = Strong
```

`mysteryFunc4 :: String -> PasswordStrength`

This function actually returns a value of the user-defined `PasswordStrength` data type and determines the strength of a given “password” string based on length, character case, etc.

11. (6 min) To Curry, or Not to Curry [Midterm review - Haskell]

a) (3 min) Write a function curr that converts a function of 1 argument which is a tuple of 3 integers into a curried function of 3 integer arguments returning an integer. Also include the function type.

Example:

```
f (x,y,z) = x + y + z
g = curr f
g 1 2 3 -- Returns 6
```

Solution:

```
curr :: ((Int, Int, Int) -> Int) -> Int -> Int -> Int -> Int
curr f = \x y z -> f (x,y,z)
```

b) (3 min) Write a function uncurr that converts a curried function of 3 integer arguments into a function that takes 1 tuple argument of size 3. Also include the function type.

Example:

```
f x y z = x + y + z
g = uncurr f
g (1,2,3) -- Returns 6
```

Solution:

```
uncurr :: (Int -> Int -> Int -> Int) -> (Int, Int, Int) -> Int
uncurr f = \(x,y,z) -> f x y z
```

12. (6 min) Flatter Flatten Tree [Midterm Review, Advanced Haskell]

Suppose we have the following Tree ADT and want to write a function that takes a `Tree` element and returns a 1D list containing all of the elements in the tree flattened in the order of a preorder traversal (this was a week 3 problem!)

```
data Tree = Empty | Node Integer [Tree]
```

Consider this potential solution that uses foldl and map to flatten the tree.

```
preOrder :: Tree -> [Integer]
preOrder (Node val children) = val : foldl (\acc lst -> acc ++ lst) (map preOrder children)
```

a) (3 min) Does this solution perform correctly for all inputs? If not, why?

This solution does not perform correctly for two reasons. First, foldl is missing the argument corresponding to the initial value of the accumulator, which should be an empty list [] in this case. Second, the pattern matching is missing the case when Tree evaluates to Empty, meaning it is non-exhaustive.

b) (3 min) Rewrite this solution to use foldl exactly once (no map/filter)

```
preOrder :: Tree -> [Integer]
preOrder Empty = []
preOrder (Node val children) =
    val : foldl (\acc elem -> acc ++ (preOrder elem)) [] children
```

13. (15 min) Immutable Inversion [Midterm Review - ADTs, Immutable Data Types]

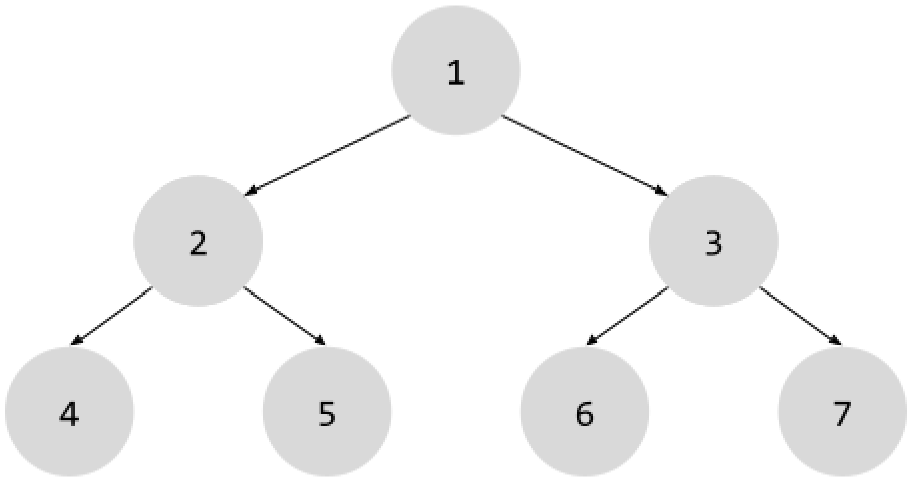
Suppose you were creating a Binary Tree `ADT` called `BTree` in Haskell. The ADT has two variants:

- a `Node`, containing an `Int` value and the left and right subtrees, and
- `Empty`, indicating the end of that branch (A `Node` whose left and right subtrees are both `Empty` is a leaf node)

a) (1 min) Show the Haskell definition for a `BTree`.

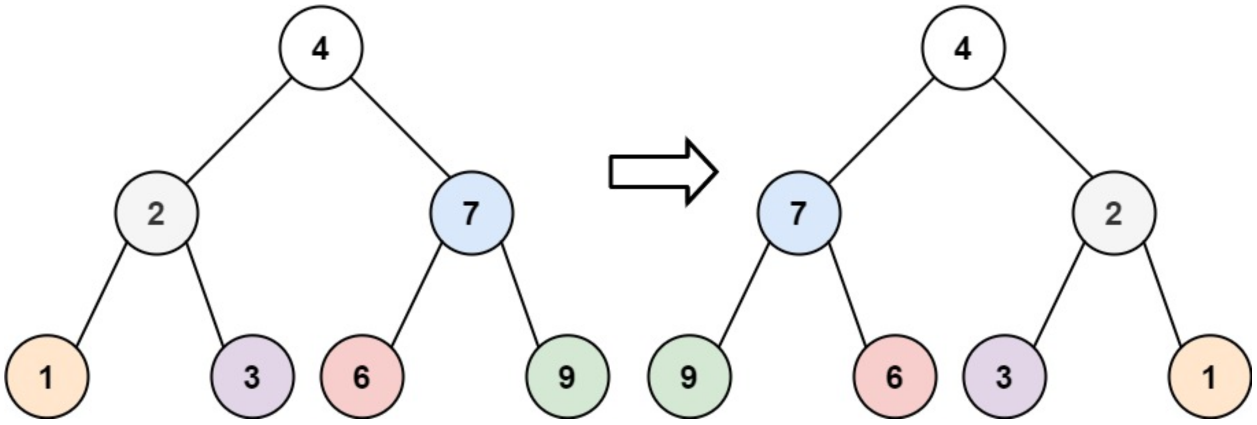
```
data BTree = Node Int BTree BTree | Empty
```

b) (4 min) Write a Haskell expression representing the following binary tree:



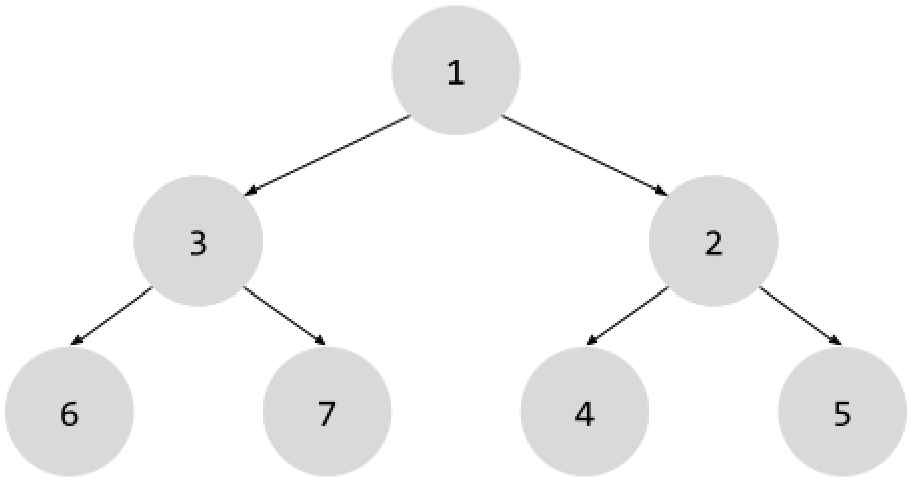
```
Node 1
    (Node 2
        (Node 4 Empty Empty)
        (Node 5 Empty Empty)
    )
    (Node 3
        (Node 6 Empty Empty)
        (Node 7 Empty Empty)
    )

-- tabs not necessary, just helpful for visualization
-- No spaces below for easy copy/paste
-- (Node 1 (Node 2 (Node 4 Empty Empty) (Node 5 Empty Empty)) (Node 3 (Node 6 Empty Empty) (Node 7 Empty
Empty)))
```

c) (8 min) Write a function `invertBTree` which takes a `BTree` and an `Int` n representing the number of levels that will be inverted (reflected). If n is greater than or equal to the depth of the tree, then invert all the levels. The root node can be considered to be level 0. An example from LeetCode has been yoinked below for an example of inverting at least 2 levels. Part d is an example of inverting less levels than the depth of the tree.



```
invertBTree :: BTree -> Int -> BTree
invertBTree Empty _ = Empty
invertBTree (Node val left right) 0 = Node val left right
invertBTree (Node val left right) n =
    Node val
        (invertBTree right (n - 1))
        (invertBTree left (n - 1))
```

d) (2 min) Let the solution to part b be assigned to the variable `btree`.
`invertBTree btree 1` produces the following tree. How many new nodes in the tree were created?



Only 3 nodes were recreated even though 6 of them are in different spots in the tree. The third pattern for invertBTree was used to create a new node with value 1, and the second pattern for invertBTree created new nodes with values 2 and 3. For nodes with values 2 and 3, the entire left and right subtrees of those were reused. (Even if their subtrees were more than 1 node, the whole thing would be reused!)

4. (12 min) [Algebraic Data Types, Immutable Data Structures]

In homework 2, you (may) have seen a question with a `Tree` algebraic data type:

```
data Tree = Empty | Node Integer [Tree]
```

a) (2 min) Adapt the Tree ADT to support nodes that can hold values of any type (like a templated class in C++). Provide the full ADT definition.

```
data Tree a = Empty | Node a [Tree a]
```

b) (10 min) Implement a function `preorder` which takes a `Tree` element and returns a 1D list containing all of the elements in the tree flattened in the order of a preorder traversal. Include a type definition. Hint: It may be helpful to use map-reduce...

Example:
`preOrder (Node 3 [(Node 2 [Node 7 []]), (Node 5 [Node 4 []])])` should return `[3, 2, 7, 5, 4]`
`preOrder (Node "3" [(Node "2" [Node "7" []]), (Node "5" [Node "4" []])])` should return `["3","2","7","5","4"]`

```
preOrder :: Tree a -> [a]
preOrder Empty = []
preOrder (Node val children) = val : foldl (\acc lst -> acc ++ lst) [] (map preOrder children)
```

5. (10 min) [Currying, Partial Application]

a) (4 min) Implement a function `sumList :: [Integer] -> Integer` using `foldl`.

Example:
`sumList [1..7]` should return `55`

```
sumList lst = foldl (\acc x -> acc + x) 0 lst
```

b) (3 min) Does your function take any inputs? Is it possible to rewrite `sumList` such that it doesn't take any input?

Yes; we can use partial function application to remove the `lst` argument.
`sumList = foldl (\acc x -> acc + x) 0`

c) (3 min) Use partial function application to remove as many function arguments as possible.

Notice that the lambda `\acc x -> acc + x` just adds two values, and is the same as just putting `(+)`. Thus we can shorten this expression even further to:
`sumList = foldl (+) 0`

Note: When we write a function without specifying the inputs, we say that it is written in "point-free" style.

6. (15 min) [Currying, Partial Application]

a) (5 min) Write a Haskell function `swap` that swaps the order of the first two arguments of a function, i.e. returns the same function but with the first two arguments swapped.

For example:
`swap div 2 4` should return `2`
`swap (-) 4 10` should return `6`
`swap (\x y z -> (x / y) + z) 3 12 6` should return `10.0`

Think simple! Fun fact: this is already defined as the function `flip`.

```
swap :: (a -> b -> c) -> b -> a -> c
swap f a b = f b a
```

b) (5 mins) Can you use the above function and PFA to write a one-liner function `filterHundred` that filters the array [1, 2, ..., 100] with the input function? e.g. you should be able to define it with the following:

```
filterHundred :: (Integer -> Bool) -> [Integer]
filterHundred =
```

For example:
`filterHundred (\x -> x `mod` 50 == 0)` should return `[50, 100]`
`filterHundred (\x -> x `div` 98 == 1)` should return `[98, 99, 100]`

```
filterHundred :: (Integer -> Bool) -> [Integer]
filterHundred = swap filter [1 .. 100]
```

c) (Challenge: 5 min) Is it possible to write a function `swapi` that takes in a number i and a function and moves the ith argument of the input function to the front? In other words, allow PFA for the ith argument. If it is possible, write it. If not, explain why.

For example:
`swapi 1 (-) 4 6` should return `2`

`swapi 3 (\x y z -> (x - y) * z) 2 10 5` should return `10`
(Instead of doing `x = 2`, `y = 10`, `z = 5`, the third argument, `z`, is now the first argument, so `z = 2, x = 10, y = 5`)

No. Consider the type of this function. The input function must take in at least as many arguments as i, while i can be as big as possible. Since Haskell is statically typed, we must know the type of the function when we declare it, so this is impossible to write.

7. (13 min) [Currying, Partial Application]

a) (2 min) Write a Haskell function called `subTwo` that takes in two integers and returns their difference (note: it should subtract the first argument from the second)

Example:
`subTwo 8 10` should return 2

```
subTwo :: Int -> Int -> Int
subTwo x y = y - x
```

b) (4 min) Using the function you just wrote, write a new function called `subTwoCurried` that takes in a single integer x and returns a function that, when called with another integer y, returns their difference (note: it should return y - x)

Given one input, we want to return a function that takes one more argument and returns the difference → we can use lambdas! This is similar to `mult3` from lecture

```
subTwoCurried :: Int -> (Int -> Int)
subTwoCurried x = \y -> y - x
```

c) (2 min) Using `subTwoCurried` write a function called `subEighteen` that represents the subtraction of an integer by 18

```
subEighteen :: Int -> Int
subEighteen = subTwoCurried 18
```

d) (5 min) Finally, create a function called `subAll` that takes in a list of integers and returns a new list where each element is decreased by 12

Because we want to perform an operation on all elements within a list, we will use the map function. We make use of currying by using the function defined in part c as the filter function for map. Finally, because of partial function application, we can also remove the list argument.

```
subAll :: [Int] -> [Int]
subAll = map (subTwoCurried 12)
```

8. (8 min) [Immutable Data Structures]
A difference between C/C++ and Python is that strings are mutable in the former, while immutable in the latter. What are the benefits to Python strings being immutable? What are the drawbacks?

Here are a couple of benefits and drawbacks, but this is definitely not a comprehensive list. One of the biggest benefits of strings being immutable is that they can be used as keys to dictionaries, since their hashes will not change. It also enables more efficient access and slicing, since we basically treat strings as const variables.

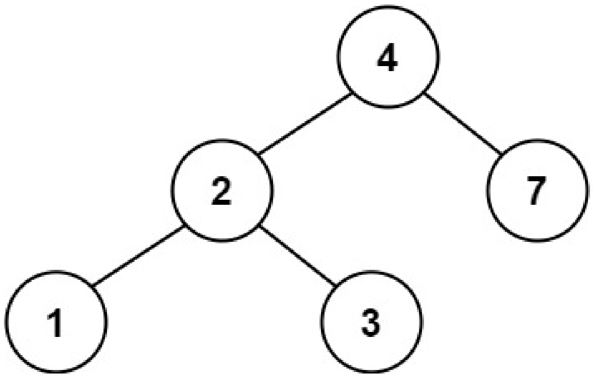
One place where mutable strings would be better is if there is an algorithm that involves concatenating or changing strings repeatedly; immutable strings means that Python needs to allocate memory for a new string each time, increasing the total memory usage. Overall, immutable strings lead to less efficiency and less flexibility in optimizing code.

9. (12 min) [Algebraic Data Types]

a) (3 min) Define a new Haskell type `BinaryTree` that has two value constructors - `EmptyTree` that takes in no parameters, and `Node` which takes in three parameters, an `Integer` value, and two `BinaryTrees` (representing the two subtrees of the current node).

```
data BinaryTree = EmptyTree | Node Integer BinaryTree BinaryTree
```

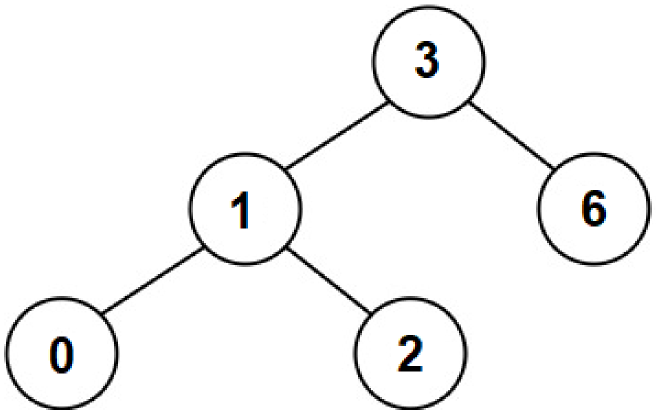
b) (2 min) Using your `BinaryTree` ADT, write an expression representing the following tree:



```
myTree = Node 4
           (Node 2
                 (Node 1 EmptyTree EmptyTree)
                 (Node 3 EmptyTree EmptyTree)
           )
             (Node 7 EmptyTree EmptyTree)
```

Split up into multiple lines for further clarity (not necessary).

c) (7 min) Write a function `decTree` that traverses through a tree and decrements the value of all nodes by 1. Be sure to include your type declaration! When run on the tree from part b, this should produce the following result:



```
decTree :: BinaryTree -> BinaryTree
decTree EmptyTree = EmptyTree
decTree (Node n left right) =
        Node (n-1) (decTree left) (decTree right)
```

2. (10 min) [Currying, Partial Application]

For each of the following, do the two type signatures represent the same function?

a) (2 min) `a -> (b -> c) -> d` and `(a -> (b -> c)) -> d`

No, the first function takes 2 parameters (`a` and `b -> c`) while the second just takes 1 parameter (`a -> (b -> c)`) before returning `d`.

b) (2 min) `a -> (b -> c) -> d` and `a -> ((b -> c) -> d)`

Yes, while the parameter counts are different (the first takes 2 and the second takes 1), the return value of the second function is yet another function which requires yet another parameter.

c) (2 min) `a -> (b -> c -> d) -> e` and `a -> (b -> (c -> d)) -> e`

Yes, the second parameter `(b -> c -> d)` is represented in curried form. Any function in a type definition, even those in the middle of the definition, can be represented in a curried fashion.

d) (2 min)
`a -> (b -> c) -> [d -> (e -> f)]` and `a -> (b -> c) -> [d -> e -> f]`

Yes, the return value of the first function is a list of functions. The second function returns a list of functions, but the type signature has been “un”-curried.

e) (2 min)
`a -> b -> c -> d -> e -> f` and `a -> (b -> (c -> (d -> (e -> f))))`

Yes, the second function is a completely curried version of the first function.

1. (10 min) [Currying, Partial Application]

a) (1 min) `addFive = (+ 5)`

```
addFive :: Int -> Int
```

b) (1 min) `weirdFilter = filter (\x -> x < 136101521 - 34551213)`

```
weirdFilter :: [Int] -> [Int]
```

c) (2 min) `yFunc = (\m b x -> m * x + b) 5 2`

```
yFunc :: Int -> Int
```

d) (3 min) `f = (\g h i -> i h g) ["Carey"] [131]`

```
f :: ([Int] -> [String] -> t) -> t
```

function `i` takes `h` and `g` as input
`g` is of type `[String]`
`h` is of type `[Int]`
PFA is applied to lambda, so expression is of type `i`
`i` takes `h` and `g` so `i :: ([Int] -> [String] -> t)` [`t` is the return type of `i`]
this is then returned by `f` so we add another `-> t`

e) (3 min)

```
bigBoi =
   (\ a b c d -> d ++ c : (map (\x -> x * b) a )) [1, 2, 3, 5, 8] 13 1
```

```
bigBoi :: [Int] -> [Int]
```

(since we already have the first 3 arguments for the lambda, we only need consider the type of d, which based on ++ must be a list)

Using `foldr`, write a function called `compress` that takes in a list of type a. If an element appears twice or more in a row, they should be replaced with a single copy of the element.

Example:
`compress "aaaabccaadeeee"` should return `"abcade"`.

Solution:
Haskell Code:

```
compress :: Eq a => [a] -> [a]
compress x = foldr(\a b -> if a == (head b) then b else a:b) [last x] x
```

3. (5 min) [Haskell Functions, Local Bindings, Control Flow]
Type Signatures
a) Translate the following type signatures into plain speech. For example, the type `[Integer] -> Bool` is a function that takes a list of integers as input and returns a boolean.

```
(a -> a -> Bool) -> Int

(b -> a -> b) -> b -> [a] -> b

(a -> b) -> ((a -> b) -> c) -> c
```

Solution:
The first is a function which takes a function as input and returns an integer. The input function has two arguments of the same type and returns a boolean.

The second is the type signature for `foldl`. It takes a function as its first argument, some accumulator as its second argument, and a list of some type as its third argument. It returns the same type as the accumulator. The input function takes the accumulator type and the list item type as input and returns the accumulator type.

The third is a function that takes two functions as input. The second function takes a function of the same type as the first function as input. The return type is the same as the return type of the second input function.

b) What are the type signatures for the following functions?

```
twice f x = f (f x)

applyAll [] x = x
applyAll (f:fs) x = applyAll fs (f x)
```

Solution
`twice :: (a -> a) -> a -> a`
`twice` accepts a function f and a value of type a. The function f takes one argument of type a and returns type a. This must be true since the result of f can be used as input to f.

`applyAll :: [(a -> a)] -> a -> a`
`applyAll` takes a list of functions that accept a single argument of type a and return a value of type a. It also takes a second parameter of type a. It returns a value of type a. We know `applyAll` must return type a because of the base case in which x is returned. We also see that the second parameter is of type a for the same reason. The tricker part is realizing that the list is a list of functions that accept type a and return type a. We can determine this using the second line in which f is applied to x and then fed as the second argument to applyAll (which we determined is of type a).
9. (8 min) [Map/Filter/Reduce (First-Class/Higher-Order Functions)]
Write a Haskell function `filterTwinPrimes` that takes a list of integers and filters out the twin prime numbers (i.e. returns a new list containing only the prime numbers whose twin also appear in the list). Twin primes are prime numbers that are 2 or less apart. For example, 17 & 19 are twin primes. We will not consider 2 and 3 to be twin primes.

Hint: `isPrime` (the function you built for problem 5) can be useful

Example:
`filterTwinPrimes [1, 2, 3, 4, 5, 6, 7, 8, 9]` should return `[3, 5, 7]`
`filterTwinPrimes [50, 49..10]` should return `[43, 41, 31, 29, 19, 17, 13, 11]`

Solution:

```
filterTwinPrimes :: [Int] -> [Int]
filterTwinPrimes numbers = foldr processNumber [] numbers
    where
        processNumber x acc
            | isPrime x = if elem (x-2) numbers && isPrime (x-2)
                            || elem (x+2) numbers && isPrime (x+2)
                then x : acc
                else acc
            | otherwise = acc
```

The main idea of this solution is to check if an element is prime AND that its possible twin (x+2 or x-2) is also in the list. If that is the case, then we add that number at the front to the accumulator.
10. (8 min) [Haskell Functions, Local Bindings, Control Flow]
a) Write a function `fib` which takes an `Int` n and returns the nth Fibonacci number (the sequence starts 1, 1, 2, 3, 5, 8, etc.). Include a function signature. You may assume n > 0.

Example: `fib 6` should return 8

Solution:

```
fib :: Int -> Int
fib 1 = 1
fib 2 = 1
fib n = fib (n - 1) + fib (n - 2)
```

b) Try running your function for large values. What's the time-complexity of your first solution? Can you write a fibonacci function that is O(N)? *Hint: You may want to use a helper function to perform a "sliding window".*

Solution:
The function has a noticeable delay once you get past ~30ish (at least on my CPU). The trivial recursive solution for `fib` is O(2^n) – every call to `fib` results in two more calls to `fib`.

But the O(N) solution is iterative, mutating a local counter to perform a loop! How can we do that with recursion? While we can't mutate variables within a function, we can mutate them between functions with an extra parameter :D.

Aside: this lets you transform a lot of "loop-based" solutions into recursion! It may be useful to keep this in the back of your mind if you can't figure out a recursive solution

```
fib2 :: Int -> Int
fib2 n =
    let
        helper i a b
            | i == n = b
            | otherwise = helper (i + 1) b (a + b)
    in helper 1 0 1
```

4. [10 points] Write a Haskell function called get_every_nth that takes in an Integer n and a list of any type that returns a sublist of every n th element. For example: get_every_nth 2 ["hi","what's up","hello"] should return ["what's up"] get_every_nth 5 [31 .. 48] should return [35, 40, 45] Your function must have a type signature and must use either filter or foldl. It must be less than 8 lines long

get_every_nth n lst = foldr (\(i, x) acc -> if i `mod` n == 0 then x : acc else acc) [] (zip [1..] lst)
-- if it is actually the nth number, the modulus should return 0, and thus, we can add it to the list, otherwise we just move on and don't concatenate
get_every_nth :: Integer -> [a] -> [a] get_every_nth n lst = map (\tup -> snd tup) -- or map (\(_,x) -> x) (filter (\tup -> (fst tup) `mod` n == 0) (zip [1..] lst))

3. [16 points-2/2/2/10] In this question, you will be implementing directed graph algebraic data types and algorithms in Haskell. Recall that a graph is composed of nodes that hold values, and edges which connect nodes to other nodes. In a directed graph, edges are unidirectional, so an outgoing edge from node A to node B does NOT imply that B also has a direct connection to A. You may assume that all graphs have at least one node. a. Show the Haskell definition for an algebraic data type called "Graph" that has a single "Node" variant. Each Node must have one Integer value, and a list of zero or more adjacent nodes that can be reached from the current node. Answer: Any of the following answers is OK: data Graph = Node Int [Graph] data Graph = Node Integer [Graph] data Graph = Node [Graph] Int data Graph = Node [Graph] Integer If the

student uses a type variable with a type of Integral that's ok too. All other answers get a zero. b. Using your Graph ADT, write down a Haskell expression that represents a graph with two nodes: 1. A node with the value 5 with no outgoing edges 2. A node with the value 42, with an outgoing edge to the previous node Answer: The students may use any variable names they like (instead of a and b). Given these definitions from part a: data Graph = Node Int [Graph] data Graph = Node Integer [Graph] Either of the following options are valid: a = Node 5 [] – 1 point b = Node 42 [a] – 1 point b = Node 42 [Node 5 []] – 2 points

Given these definitions from part a: data Graph = Node [Graph] Int data Graph = Node [Graph] Integer Either of the following options are valid: a = Node [] 5 – 1 point b = Node [a] 42 – 1 point b = Node [Node [] 5] 42 – 2 points c. Carey has designed his own graph ADT, and written a Haskell function that adds a node to the "front" of an existing graph g that is passed in as a parameter: add_to_front g = Node 0 [g] In Haskell, creating new data structures incurs cost. Assuming there are N nodes in the existing graph g, what is the time complexity of Carey's function (i.e., the big-O)? Why? Answer: O(1) because only a single new node is being created, and it links to the original graph. The existing graph need not be regenerated in any way. d. Write a function called sum_of_graph that takes in one argument (a Graph) and returns the sum of all nodes in the graph. Your function must include a type annotation for full credit. You may assume that the passed-in graph is guaranteed to have no cycles. You may have helper function(s). Your solution MUST be shorter than 8 lines long.

--- matt's solution (map + sum) sum_of_graph (Node val edges) = val + sum (map sum_of_graph edges)
--- matt's solution (map + fold) sum_of_graph (Node val edges) = foldl (+) val (map sum_of_graph edges)

7. [9 points-3/3/3] We know that int and float are usually not subtypes of each other. Assume you have a programming language where you can represent integers and floating-point values with infinite precision (call the data types Integer and Float). In this language, the operations on Integer are +, -, *, / and % and the operations on Float are +, -, *, and /. a. In this language, will Integer be a subtype of Float? Why or why not? Answer: 3 points for the following: Yes, Int is a subtype of Float because: 1. Every Integer value can be represented by a Float (with the fractional part being zero). 2. Every operation on Float can also be performed on an Integer. 3 points off if they do not say "Yes" 1 point off for missing either requirement b. In this language, will Float be a subtype of Integer? Why or why not? Answer: 3 points for the following: No, Float is a NOT a subtype of Integer because: 1. Not every Float value can be represented by an Integer 2. Not every operation on an Integer can also be performed on a Float. 3 points off if they do not say "No" 2 points off for missing justification (must mention one of the two) c. If you discovered that the Float type also supports an additional operator, exponentiation, what effect would that have on your answers to a and b? Why? Answer: 3 points for the following: Neither would be subtypes of each other. Both reasons needed for full credit: 1. Floats have an operator that Integers don't support so Integers can't be a subtype of Float 2. Integers have an operator that Floats don't support so Floats can't be a subtype of Integer (or alternatively, not every float value can be an Integer)

1. [10 points-2 each] For each of the items below, write a Haskell type signature for the expression. If the item is a function then use the standard haskell type notation for a function. If the item is an expression, write the type of the expression (e.g., [Int]). For any answers that use one or more type variables where concrete types can't be inferred, use "t1", "t2", etc. for your type variable names.
a. foo bar = map (\x -> 2 * x) bar
b. z = \x -> x ++ ["carey"]
c. m = [((\a -> length a < 7) x,take 3 y) | x <- ["matt","ashwin","siddarth","ruining"], y <- ["monday","tuesday","wednesday","thursday","friday"]]
d. z = foldl (\x y -> x ++ y)
e. foo bar = []

1.
a. foo :: [Int] -> [Int]
b. y :: [[char]] -> [[char]]

c. m :: [(String, [String])]
SOLUTION: m :: [(Bool, String)]
d. z :: [t1] -> [t1] -> []
SOLUTION: z :: [t1] -> [[t1]] -> [t1]
If the accumulator is already a list, then the actual list it traverses has to be a list of lists
e. foo :: [t1] -> []
SOLUTION:
[t1] -> [t2]

Problem #8: 15 points a. Give 2 points for a proper type signature - the type signature is only required for the top-level substr function. Zero points for anything else. Any of these variations is OK: substr :: String -> String -> Bool substr :: String -> (String -> Bool) substr :: [Char] -> [Char] -> Bool substr :: [Char] -> ([Char] -> Bool) b. Here's one possible solution: MT Variation #1 substr [] [] = True substr _ [] = False substr xs (y:ys) = (substr xs ys) || (front_substr xs (y:ys)) front_substr :: String -> String -> Bool -- Type signature here not required for full points front_substr [] _ = True front_substr (x:xs) (y:ys) | x == y = front_substr xs ys | otherwise = False